新人技術者のための

ロジカル・シンキング入門 第9回

モンスターはどこへ消えた？…最適化設計（その2）

冴木 元

LOGICAL THINKING

前回に引き続き，組み込みシステム開発における最適化設計について解説する．今回は，メモリのアクセス速度がボトル・ネックとなる場合の最適化について解説する．また，最適化の際に新たなバグを混入させないためのテスト方法について解説する．

（編集部）

前回（本誌2007年2月号，pp.157-163）3Dグラフィックス・モジュールの最適化を迫られたFさんでしたが，なんとか処理速度を向上することに成功したようです．とりあえず，修正済みのライブラリ・モジュールをリリースして様子を見てもらうことにしました．

画面の動きは前より速くなり，これならば一応許容範囲ではないかという評価を受けたFさんたちは，まずまずの結果にほっと胸をなでおろしました．

しかし，システム・テストが進むにつれ，新たな問題が生じました．あるゲームで不具合が発生したというのです．3面までゲームが進むと新たに登場してくるモンスターがあるのですが，これが時々消えてなくなるというのです．再現してみると，モンスターが壁にぶつかったとたんにみるみる形が崩れていくのが分かりました．そして切り分けの結果，まずいことにFさんたちの3Dグラフィックス・モジュールのバグであることが分かってしまいました．最適化した結果，新たなバグを入れてしまったようなのです．

問題はこれだけではありませんでした．処理速度は以前よりも向上したように思えたのですが，ある描画パターンになると，以前と同じように描画が滞るのです．これも切り分けの結果，3Dグラフィックス・モジュールが処理速度のボトル・ネックになっていることが分かりました．

しばらく見なかったはずのトロトロとした描画を見ながら，Fさんたちはため息をもらすのでした．「バグを生んだ上に，まだ最適化する必要があるのか…」

Fさんのケースに見られるように，最適化が新たなバグを生んでしまうことはよくあります．前回も述べたように，最適化というのはあくまでも製品が仕様通り動くことが前提であるため，最適化の結果，仕様を満たさなくなってしまうのでは元も子もありません．また，最適化というのは「何を基準にするか」というのも難しい問題となります．普通は処理速度が最悪になるデータを選んで，これを基準に最適化の度合いを測っていきます．しかし，後からもっと処理速度が遅くなるデータが見つかってしまう，ということはしばしば起こり得ます．最適化には「既に作ったものをいかに守るか」という観点も落とせないのです．

さて，前回はCPUの演算量がボトル・ネックになるケースを取り上げました．今回は，メモリ速度がボトル・ネックになるケースを取り上げます．また，冒頭のエピソードのようなバグを防ぐ方法についても考えてみたいと思います．

● 外部メモリの巨大なデータにアクセスしたいのだが…

画像データのように大きなデータは，CPUの内部メモリに置くことができないので，通常，外部メモリに配置します．ここで外部メモリにアクセスするには，大抵の場合1クロックではすまないため，CPUが外部メモリに直接アクセスする構成にしてしまうと，処理速度が著しく遅くなっ

KeyWord 最適化設計，データ・キャッシュ，DMA，パイプライン処理，テスト自動化，レビュー，エビデンス

図1
メモリ速度がボトル・ネックとなるケースと対策
処理データが大きすぎてCPUの内部メモリに乗りきらない場合（画像処理などに多い），データを外部メモリに置くのだが，外部メモリに直接CPUがアクセスすると処理が遅くなる．そこで，DMAなどを用いて，データをいったん内部メモリに転送して処理する．

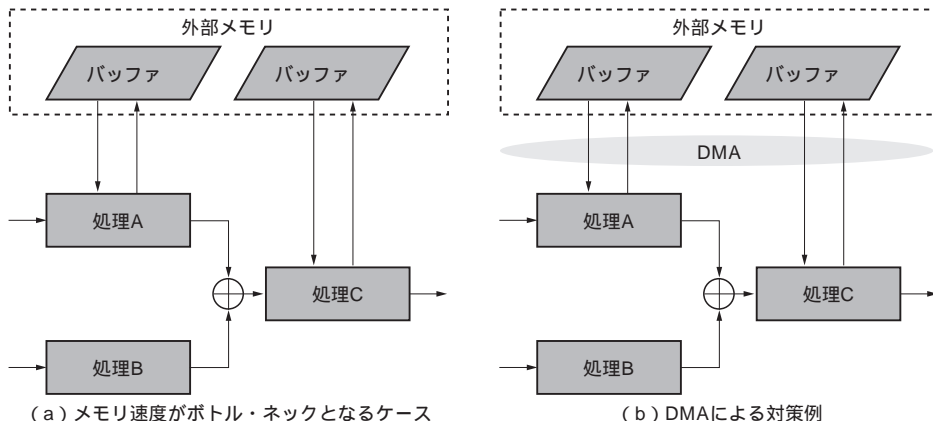
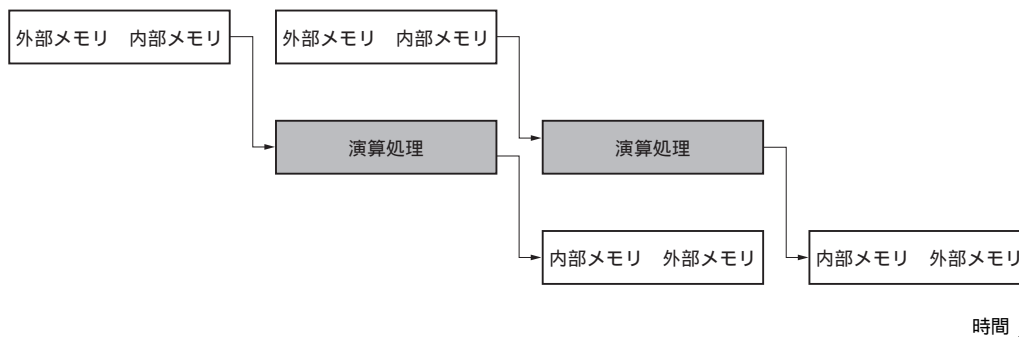



図2
転送と演算の並列化
処理データが大きい場合の解決策として，演算そのものは内部メモリに置いて，演算の前後で外部メモリと内部メモリとの間で転送を行う方法がある．
　転送処理にはCPUではなく，DMAを用いる．いわば，データ・キャッシュが行う処理をプログラマが手書きするイメージ．

てしまいます［**図1（a）**］．

「いや，データ・キャッシュがCPUに実装されていれば問題はないはずだ」と考える方もいらっしゃるかもしれません．確かに，データ・キャッシュが実装されていれば，外部メモリへのCPUアクセスはキャッシュに対するアクセスに変わるため，処理速度の問題は生じないとも考えられそうです．しかし，実際には巨大なデータを外部メモリに置いて処理を進める場合，データ・キャッシュの機能に頼るだけでは最適化が不十分な場合が多くあるのです．

要するに，「最適化」と一言で言っても，さまざまなアプローチが必要になるということです．前回に紹介した最適化のテクニックは，CPUの命令を効果的に使いこなしてコーディングすることで処理速度の向上を図るものでした．しかし，今ここで例に挙げているようなケースでは，CPUの命令の使い方を工夫しても最適化は図れません．では，どのような解決策が考えられるでしょうか．

● DMA転送を活用する

外部メモリにデータを置いていることが処理速度のネックになる場合，DMA（direct memory access）を使って外部メモリと内部メモリとの間で転送を行う方法が考えられます［**図1（b）**］．すなわち，

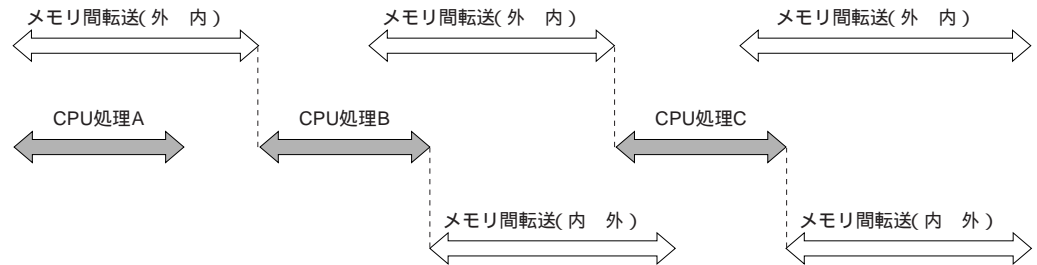
1）DMA転送（外部メモリ→内部メモリ）
2）CPUによる処理（内部メモリにアクセス）
3）DMA転送（内部メモリ→外部メモリ）

というステップを踏むわけです．

DMAを用いる理由の一つは，転送そのものにCPUの負担をかけないためです．メモリ間のデータ転送をCPUで行わなければ，その分だけCPUは画像データそのものの処理に集中できます．しかし，DMAによる転送を加えただけでは，まだ最適化は不十分です．なぜなら，DMAの転送中の空き時間をCPUが有効利用していないからです．

● 転送のパイプライン化で対応

CPU処理とDMA転送は別々のハードウェアを利用するので，当然同時に動作させられます．従って，DMA転送中にCPUは別の処理を行えます．この「DMA転送中の空き時間をうまく活用する」というところに，最適化の余地

図3
転送のタイミング・チャート
DMA転送とCPU処理をうまく最適化するためには，転送のタイム・チャートを検討し，互いに依存する転送と処理は前後関係が矛盾しないようにうまくパイプラインを組む必要がある．これはプログラム・フロー全体を見通す必要がある作業である．

が残されています．すなわち，DMA転送とCPU処理をうまくパイプライン化することで，初めて十分に最適化できたと言えます（**図2**）．

具体的には，外部メモリ－内部メモリ間のDMA転送を，CPUによる処理の前後に置きます．DMAで外部メモリから内部メモリに転送したデータをCPUで処理し，それを再びDMAで内部メモリから外部メモリへと書き戻す必要があります．この転送（DMA）→処理（CPU）→転送（DMA）という一連の流れを矛盾なく並列に処理できるようにすれば，大きな最適化の効果が見込めます．

ここまでに述べた最適化のポイントを**表1**にまとめます．

表1　メモリ速度がボトル・ネックになるモジュールの最適化

最適化設計の方法	対処方法
DMAの活用	DMAにより，内部メモリと外部メモリとの間でデータを転送し，演算処理をパイプライン化する
データ・キャッシュ	データ・キャッシュを用いる（補足的なものにとどまる）

● フロー全体を見渡した最適化が必要

ところで，パイプライン化というのは概念としてはシンプルですが，実際にコーディングに反映するとなると，なかなか難しいものとなります．

プログラマはまず，DMA転送とCPU処理のタイム・チャートを調べ，両者をうまく配置できるような設計を考えなければなりません（**図3**）．これは，

1）DMAとCPUがなるべく同時処理を行えるように配置する
2）互いに依存する処理は前後関係を正しく配置する

という二つの要請をともに満たすように，DMA転送とCPU処理を配置していく作業です．

同時処理を行えるように配置されていなければ，処理速度の十分な向上は見込めません．かといって，互いに依存する処理の前後関係が矛盾していると，処理済みのデータを上書きしてしまったり，内部メモリで行った処理が外部メモリに反映されなかったり，といったことが生じてしまいます．ここを見落とすと，冒頭の例で挙げた「モンスターが壁にぶつかると崩れてしまう」といったような，最適化によるバグ混入を生んでしまうのです．

やっかいなのは，このパイプライン化というのが，プログラム・フロー全体を見通した上で行わなければならない作業であることです．

前回に挙げたようなアセンブリ・コードによる最適化は，最適化の対象となる関数の中に閉じた作業です．従って，その関数が単体で最適化後も正しく動作することさえ確認できれば，その最適化がほかの個所に影響を及ぼす心配はないと言えます．

もちろん，アセンブリ・コードによる最適化自体は極めて難易度の高いものです．しかし，最適化後の動作確認はそれほど難しくないと言えるでしょう．なぜなら，アセンブリ・コードによる最適化は，既に述べたように，常に関数単体に閉じて行うことができるからです．従って，最適化した関数が完ぺきならば，残りのブロックが影響を受けることはありません．

ところが，ここで述べているパイプライン化というのは，プログラム・フロー全体を見通す必要があるものです．従って，最適化が十分になされているか，最適化によるバグが生じないかを，常にプログラム全体を見通して確認する必要があります．

●「全体を見通す」という意味：Wait抜けの例

ここで，プログラム・フロー全体を見通す難しさについて，DMAのウェイト（wait）処理を例に説明してみたいと思います．DMAの転送は大きく分けて，次の3段階から成ります（**図4**）．

図4
DMAのウェイト抜け
DMAの転送は大きく分けて，1）転送パラメータの設定，2）転送開始，3）ウェイト処理，の3段階から成る．最後のウェイト処理を抜かしてしまうと，次のDMA転送時にハングアップする原因となる．これは単純ミスのようでいて，実際のコーディングでは防ぐのがなかなか難しい．

図6　工程と工数分布の実際
概念的には，工程は一つ一つ実施することになっている．しかし最適化では，通常，作っては確認することを繰り返す．この場合，テストにかける工数をしかるべきタイミングで充実させ，新たなバグを生まないようにする工夫が必要となる．

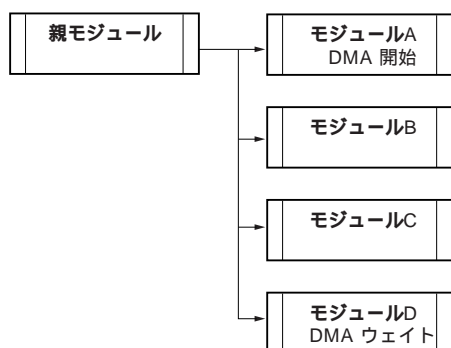
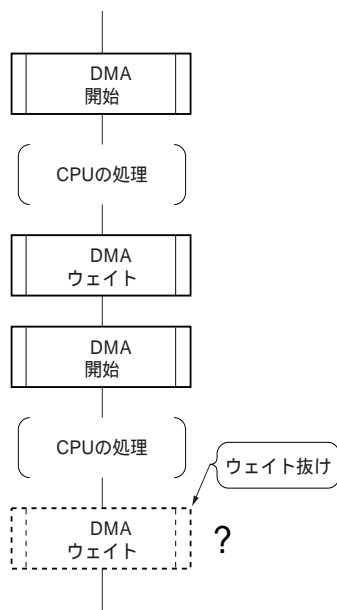



図5　フロー全体を考えた最適化
パイプライン化による最適化を突き詰めると，コードのフロー全体にまたがった最適化が必要となってくる．この中でCPUの処理とDMA転送を最適に配置し，かつウェイト抜けのようなミスを防ぐのは難易度が高い．

1）DMAの転送パラメータの設定
2）DMA転送開始
3）ウェイト処理

このウェイト処理とは，DMAのレジスタに用意されている転送終了を知らせるフラグをCPUが見張っている処理のことです．DMAには，転送処理が終了すると反転するようなフラグがレジスタに必ず用意されています．そのため，CPUはこれを見張ることでDMAの転送終了を知ることができます．

従って，最後のウェイト処理を抜かしてしまうと，正しい転送シーケンスとは言えなくなります．また，次のDMA転送時にハングアップしてしまう原因となります．

この「ウェイト処理の抜け」というのは，一見単純ミスのようでいて，実際のコーディングでは防ぐのがなかなか難しいものです．なぜなら，**2）**DMA転送開始と**3）**ウェイト処理はプログラマが一目で見渡せる個所に記述されるとは限らないからです．パイプライン化による最適化をギリギリまで推し進めると，例えば，**2）**DMA転送開始と**3）**ウェイト処理が異なった関数の中に記述されることも起こり得ます（**図5**）．

従って，プログラマはDMA転送のタイミング・チャート（**図3**）を考える設計の段階から，ウェイト抜けなどのミスが生じないように注意しなければなりません．そして，実際のコーディングでも，ウェイト処理の記述に抜けはないかという観点からソース・コード・レビューを行う必要があるでしょう．また，最適化後もミスがなかったかどうかを確認するために，適切なテスト項目が用意されている必要があります．すなわち，設計，コーディング，テストの各工程において，注意深い確認が必要になります．

● 最適化によるバグに注意

何度も言うようですが，モジュールが仕様通り正しく動くということが最適化の前提です．従って，最適化の工程においては最適化作業そのものと並行して，新たなバグの発生を防ぐためのテスト・ケースを用意する必要があります．

一般的なソフトウェア工学の概念によれば，「設計」，「コーディング」，「テスト」は別々の工程であり，一つの工程が終わってから次の工程が始まることになっています．し

かし，最適化の作業においては，これらのサイクルを短い期間に何度も回す必要が生じます．つまり「コーディングを少し進めたらテストする」を繰り返すことで，品質の劣化を防ぐことになります（**図6**）．

● テストは時間がかかる

ここで問題となってくるのが，テストそれ自体にかかる工数の増大です．本来，組み込みシステム開発に限らず，開発した各モジュールは十分にテストしてから結合する必要があります．しかし，テストというのは，工数不足を理由にしばしば省略されることが多い工程でもあります．これは「テスト項目そのものが用意されていない」というような極端なケースばかりとは限りません[1]．開発の初期においてはきちんとテスト項目を用意して，テストしてからモジュールをリリースしていたような場合でも，開発の終期にさしかかるとテストを省略してしまうこともあり得ます．なぜなら，モジュールの開発終期というのは，わずかな変更（バグの修正であれ，仕様変更であれ）を加えて短期間でリリースしてほしい，と要求されるケースが多々あるからです．「修正はごくわずかだから，まさか問題は生じないだろう」と考えて未検証のモジュールを提供してしまい，後で大問題となるケースは枚挙にいとまがありません．

● テスト実施を自動化して対応

テスト・ケースを実施するのにも時間がかかるため，テストの効率化も実は必要となってきます．テストの効率化というと，まず，

● 不要なテスト項目を省いて工数削減

というアプローチが考えられます．しかし，効率化の手段がこれだけしかない場合は問題です．なぜなら，このアプローチはしばしばテスト項目の省きすぎにつながるからです．差し迫った開発工数の中で「多分大丈夫なはずだ」と考えて，テストを大幅に省略してしまう誘惑に勝つのは容易ではありません．従って，

● テストの自動化による工数削減

という別のアプローチが有効だと考えられます．具体的には，テストそのものをプログラム化してしまうことでテスト工程を自動化し，テストにかかる工数を少なくしながら，テストを確実に実施できるようにするわけです（**図7**）．

組み込みモジュールの場合，リファレンス・モデルが出力する期待値との一致をもってテストの合否を判定するケースはよくあります[2]．このような場合，テスト・プログラムが動作パラメータを受け付けられるようにしておけば，パラメータの組み合わせをスクリプトでテスト・プログラムに与えられます．こうすると，何千通りのテスト・パターンがある場合でも，テストそのものはスクリプトで自動化されているので，人間の作業は新しいモジュールをテスト環境に結合する作業だけとなります．スクリプトの動作時間は長いかもしれませんが，十分なCPU速度を備えたパソコンを用意するなどによって短縮できるでしょう．テスト・プログラムを夜間に動作させるようにすれば，わずかな修正→終夜運転による再テスト→翌日リリース，というステップを踏むことができます．この方法なら，テスト項目をあれこれ削ってテスト工数を削減するよりも確実な品質が見込めます．

テスト・
プログラム

モジュール
Version 1.0

モジュール
Version 1.1

テストを自動化し，
改版時に適用

モジュール
Version 1.2

図7　テスト実施の自動化
最適化作業中の品質劣化を防ぐためには，適切なテストを効率良く実施することが不可欠である．よく用いられるのは，テストそのものをプログラムによって自動化してしまう方法である．もちろん，テスト・ケースそのものが適切に工夫されていることが前提となる．

● テスト・ケースの中身が大事

もちろん，実施するテスト・ケースそのものに無駄や漏れがあっては，いくらテストを自動化しても品質の劣化は防げません．自動化の前提には，テスト・ケースが適切に確立されていることが前提となります．テスト・ケースが不十分であったり，偏りがあって的確にバグが洗い出せないものであった場合には，ただやみくもにテストを自動化したとしても，バグ混入は避けられないでしょう．テストは，実施方法が自動化されている，いないにかかわらず，まずテスト・ケースが適切に組み立てられていることが重要なのです．

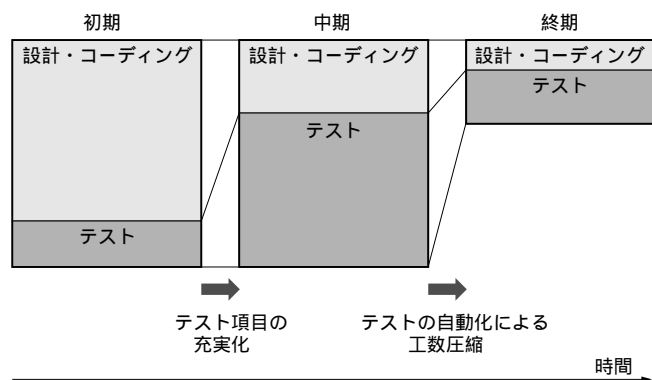



図8　最適化設計に必要な工程の組み立て
初期段階では，設計とコーディングに時間がかかるのでテストは軽めとなる．しかし，ある段階でテスト項目を充実させて品質を確かなものにすることが必要．終期にはテストの自動化で工数の削減とテストの充実を両立させる．

●「工程」について．その理論と実際

最後に，「工程」という考え方から見た最適化設計とテストについて述べておきたいことがあります．

最適化が1度のコーディングで終了することはまれです．大抵の場合，試行錯誤を経て，徐々に処理スピードやメモリ使用量の目標値に近づけていくことになります．

この場合，コーディング→テストという工程を杓子定規にとらえて「最適化コーディングという工程の終了判定が確実に行えない限り，次工程であるテスト工程は開始できない」などと考え，一切テストを行わずに最適化を進めたら大変なことになります．なぜなら，最適化コーディングをすべて終わらせてからテストして，もしバグが見つかったら，どの変更でミスが混入したか分からなくなってしまう恐れがあるからです．そうなると，せっかく最適化したソース・コードをまるごと捨てることにもなりかねません．最適化に限らず，ソースの変更・修正は「少しずつ確実に」が原則です．

世にある開発標準のたぐいは，よく考えて使えば品質保証に役立つことも少なくないのですが，反面，単純に杓子定規にあてはめるとかえって混乱が増すものも数多くあります．規則を作るときは趣旨をよく考えることが大切だと，筆者は考えます(3)．

● テスト項目の充実と高速化を両立させる

ここまで述べてきたコーディングとテストを繰り返すような開発スタイルを図にすると，**図8**のようになります．「初期」，「中期」，「終期」と時間を分けてありますが，この三つを厳密に区別する必要があるわけではありません．ポイントはむしろ，「テスト項目の充実化」，「テストの自動化・高速化による工数の圧縮」という二つのアプローチをともに満たす必要があるという点にあります．テストを自動化してテスト実施にかかる時間を短縮する一方，テスト項目は必要なだけ充実させてコーディングとテストを繰り返す体制を整えることで，より確実に品質を保証できます．

初めて開発する組み込みモジュールの場合，仕様がよくわかっていないため，テスト項目を考えるのに一苦労することがよくあります．これまでに存在しないアルゴリズムを開発対象としたような場合は特にそうです．その場合，最初は設計とコーディングだけで手一杯になってしまうことがあります．

しかし，そのような場合でも，ある段階でテスト項目を充実させて品質保証を確かなものにする，というフェーズが必要です（**図8**の①）．なぜなら，作りっぱなしでリリースしてしまえば，当然バグが潜んでいる恐れがあるからです．テストをろくにせずに品質の低いモジュールを提供し続けていると，「あそこのチームは工程全体をストップさせる常習犯だ」とみなされてしまうことにもつながりかねません．

一方で，テスト項目の数が充実してくると，当然テストそのものの実施に時間がかかることになります．そのため，開発が終盤にさしかかって，わずかな修正/変更とリリースを短期間で繰り返すフェーズになると，テストがしばしば省略されがちになります．これがバグの原因となることは既に述べました．テストを省略せずにテスト工数を削減して効率化するための手段として，テストの自動化が挙げられるわけです（**図8**の②）．もちろん，①と②を同時に実現できるのであれば，それに越したことはありません．テスト項目を増やすと同時に，それを自動的に実施できるようなテスト・プログラムを作っておけば手間が省けます．

● 手続き的な要請との調和

以上のような開発を進める場合でも，職場の開発方針からくる手続き的な要請から，どうしてもエビデンス（監査証跡）を残さなければならない場合もあるでしょう．「手続き的な要請」というのは，ISO 9000の認証を受けた事業所などで，「工程終了判定」の類を設けなければならない場合を指します．その場合はどうしたらよいでしょうか？

手続き的な要請と開発現場の方針を調和させる方法とし

て，以下のような方法が採り得ると思います．

まず，コーディング→テスト→コーディング→テストと試行錯誤を繰り返す開発上の要請があれば，まずそれを前提として，エビデンスの揃え方を考えるべきでしょう（**図9**）．試行錯誤を繰り返す開発の場合でも，製品をリリースする前に最終的なエビデンスは用意できるはずです．例えば，リリース前の適切な段階で最適化したソース・コードのレビューを行い，テスト結果をドキュメント化して両者を記録に残すことは可能なはずですし，それはむしろ必要だと思います．

レビューの主な目的が「誤りの発見」であることは言うまでもありません．最適化というのは，とかく個人の力量に頼るところが大きくなります．製品をリリースする前に一度もチーム単位でのレビューが行われないとしたら，満足のゆく品質保証はできません．なぜなら，最適化した個人がミスを犯したら，そのまま開発チームの外にバグのある製品が出て行く体制を作りかねないからです．

また，レビューはしばしば「新人の教育」という目的も兼ねることがあります．製品開発に必要なテクニックが文字になって書店などに並んでいるというのは，実は極めてまれです．大抵の開発テクニックというのは開発現場で培われているものであり，個人に属するところが大きいものです．これらは結局「見て覚える」，「盗んで覚える」という学習方法をとることが必然的に多くなります．開発経験の浅い人にとっては，他の経験豊かなエンジニアの仕事の結果を見ることで得るところが少なくありません．レビューを全く行わないというのであれば，この貴重な教育の機会を失ってしまうことにもなりかねません．

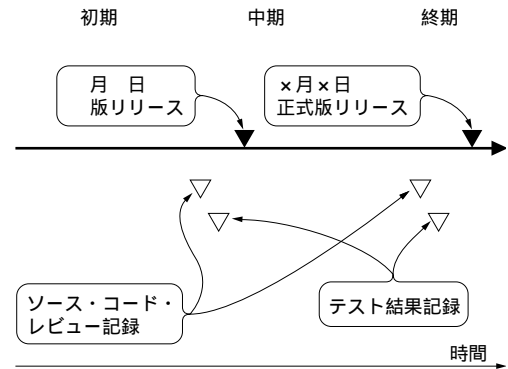



図9　最適化設計とそのエビデンス
試行錯誤を繰り返すのが前提の開発でも，リリース前には一定の記録を用意することが可能なはずだし，その必要性はむしろ大きい．製品を出す前にチーム単位でレビューする機会をなくしてしまうと，品質が保ちにくい上，開発テクニックの共有化が図れなくなってしまう．

参考・引用*文献
(1) 冴木 元；「ひたすら流すだけ」にさようなら，Design Wave Magazine，2007年1月号，pp.119-124，CQ出版社．
(2) 冴木 元；いかにしてバグの原因を突き止めるか，Design Wave Magazine，2006年6月号，pp.50-57，CQ出版社．
(3) 冴木 元；リリースしたらテーブルが消えた？，Design Wave Magazine，2006年10月号，pp.132-138，CQ出版社．

さえき はじめ

<筆者プロフィール>
冴木 元．システム・エンジニア．連載をしばらく続けていると，ネタが少なくなるころに読みたい本がいくつか出てくるものだ（いずれもソフトウェアとは無関係なものなのだが）．本屋をうろついていると，いくつかの本が手招きしているような気がする．長めの休暇が得られたら，久々に読書に充ててみようかと思う．